To appear in the International Journal of High-Speed Computing

PARALLEL INTEGER SORTING WITH MEDIUM AND FINE-SCALE PARALLELISM

LEONARDO DAGUM
Computer Sciences Corp.
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

Two new parallel integer sorting algorithms, *queue-sort* and *barrel-sort*, are presented and analyzed in detail. These algorithms do not have optimal parallel complexity, yet they show very good performance in practice. *Queue-sort* is designed for fine-scale parallel architectures which allow the queueing of multiple messages to the same destination. *Barrel-sort* is designed for medium-scale parallel architectures with a high message passing overhead. The performance results from the implementation of *queue-sort* on a Connection Machine CM-2 and *barrel-sort* on a 128 processor iPSC/860 are given. The two implementations are found to be comparable in performance but not as good as a fully vectorized bucket sort on the Cray YMP.

Keywords: parallel sorting, Connection Machine, SIMD, MIMD, distributed memory

1. Introduction. Integer sorting is a subclass of general sorting which exists when the keys to be sorted are integer in value. If the keys to be sorted are allowed *any* value then the lower-bound sequential complexity [13] to sort n keys is $O(n \log n)$. However, when the keys are restricted to be integers in the range $[1,n]$, the lower-bound sequential complexity [13] to sort n keys is only $O(n)$. The problem of integer sorting is particularly important in Monte Carlo simulations. For this reason it has been selected as one of the kernel benchmarks used to evaluate parallel supercomputers for the Numerical Aerodynamic Simulation (NAS) program at NASA [3].

The *bucket sort* algorithm [1] (or *distribution counting* [13]) achieves the lower-bound O(n) time for sequential integer sorting. On a parallel machine, the performance bounds are limited by processors as well as time. Therefore the performance bound of parallel algorithms must be measured as the product of the processor bound, P, and the time bound, T. A parallel algorithm is *optimal* if its performance bound PT is equal to the sequential time bound, T_s, for the problem. Several optimal parallel integer sorting algorithms have been proposed (see [14, 9]). However these algorithms have proved unsuitable for implementation on single instruction multiple data (SIMD) or multiple instruction multiple data (MIMD) distributed memory machines like the Connection Machine CM-2 or the Intel iPSC/860. This paper presents two parallel integer sorting algorithms which, although not optimal, have been implemented and shown to give good performance on these machines. Some theoretical analysis of these algorithms is presented, however the algorithms of this paper were borne out from an applications oriented perspective and emphasis is given to the application analysis.

The remainder of this section defines key terms used throughout the paper. Section 2 overviews the two machine models used. Sections 3 and 4 introduce and analyze the two sorting schemes, namely *queue-sort* and *barrel-sort*, respectively. Performance results are compared and discussed in section 5 and conclusions are presented in section 6.

1.1. Some Definitions. A sequence of keys, $\{K_i | i = 0, 1, \ldots, N-1\}$, will be said to be *sorted* if it is arranged in non-descending order, i.e. $K_i \leq K_{i+1} \leq K_{i+2} \ldots$. The *rank* of a particular key in a sequence is the index value i that the key would have if the sequence of keys were sorted. *Ranking*, then, is the process of arriving at a rank for all the keys in a sequence. *Sorting* is the process of permuting the keys in a sequence to produce a sorted sequence. If an initially unsorted sequence, $K_0, K_1, \ldots, K_{N-1}$ has ranks $r(0), r(1), \ldots, r(N-1)$, the sequence becomes sorted when it is rearranged in the order $K_{r(0)}, K_{r(1)}, \ldots, K_{r(N-1)}$. Sorting is said to be *stable* if equal keys retain their original relative order. In other words, a sort is stable only if $r(i) < r(j)$ whenever $K_{r(i)} = K_{r(j)}$ and $i < j$. The algorithms presented here are not stable. *Key density* refers to the number of equal keys in a sequence. The *prefix sum* of a sequence is the sequence obtained as the running sum of the original sequence elements. The j^{th} element of the prefix sum of sequence K_i is given by $K_j = \sum_0^j K_i$. Prefix operations are also referred to as *scan* operations [5]. A scan operation with binary operator $\oplus$ across an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ returns the ordered set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$. All logarithms are to base 2 unless otherwise indicated.

2. Machine Models. The algorithms presented here were implemented on two different parallel machines at NASA Ames, the Thinking Machines Connection Machine CM-2 and the Intel iPSC/860. The architectures are briefly described below.

2.1. Connection Machine. The CM-2 is a massively parallel SIMD computer consisting of many thousands of bit serial data processors under the direction of a front end computer. The system at NASA Ames consists of 32768 bit serial processors each with with 1 Mbit of memory and operating at 7 MHz. The processors and memory are packaged as 16 in a chip. Each chip also contains the routing circuitry which allows any processor to send and receive messages from any other processor in the system. In addition, there are 1024 64-bit Weitek floating point processors which are fed from the bit serial processors through a special purpose "Sprint" chip.

The Connection Machine CM-2 can be viewed two ways, either as an 11-dimensional hypercube connecting the 2048 CM chips or a 10-dimensional hypercube connecting the 1024 processing elements. The first view is the "fieldwise" model of the machine which has existed since its introduction. This view admits to the existence of at least 32768 physical processors (when using the whole machine) each storing data in fields within its local memory. The second is the more recent "slicewise" model of the machine which admits to only 1024 processing elements (when using the whole machine) each storing data in slices of 32 bits distributed across the 32 processors in the processing element. Both models allow for "virtual processing", where the resources of a single data processor may be divided to allow a greater number of virtual processors.

Regardless of the machine model, the architecture allows interprocessor communication to proceed in three manners. For very general communication with no regular pattern, the router determines the destination of messages at run time and directs the messages accordingly. This is referred to as general router communication. For communication with an irregular but static pattern, the message paths may be pre-compiled and the router will direct messages according to the pre-compiled paths. This is referred to as compiled communication and can be 5 times faster than general router communication. Finally, for communication which is perfectly regular and involves only shifts along grid axes, the system software optimizes the data layout by ensuring strictly nearest neighbor communication and uses its own pre-compiled paths. This is referred to as NEWS (for "NorthEastWest-South") communication. Despite the name, NEWS communication is not restricted to 2-dimensional grids and up to 31-dimensional NEWS grids may be specified. NEWS communication is the fastest.

The Connection Machine's processors are used only to store data. The program instructions are stored on a front end computer which also carries out any scalar computations. Instructions are sequenced from the front end to the CM through one or more sequencers. Each sequencer broadcasts instructions to 8192 processors and can execute either independent of other sequencers or combined in two or four.

2.2. Intel iPSC/860. The Intel iPSC/860 (also known as Touchstone Gamma System) is based on the new 64-bit, 40 MHz i860 microprocessor by Intel. A single node of the iPSC/860 system consists of the i860, 8 MB dynamic random access memory, and hardware for communication to other nodes. The system installed at NASA Ames consists of 128 computational nodes arranged in a seven dimensional hypercube using the direct connect routing module and the hypercube interconnect technology of the earlier, 80386-based iPSC/2. The point to point aggregate bandwidth is 2.8 MB/sec per channel and the latency for the message passing is about 149 μs for message lengths over 100 bytes (see [8]).

Interprocessor communication proceeds through the `send` and `receive` system calls. Any processor can send a message to any other processor, however the destination processor does not acquire the message unless it issues a `receive`. The high communication overhead is a result of having a software implementation of the message passing protocols.

The complete system is controlled by a system resource module (SRM), which is based on an Intel 80386 processor. This system handles compilation and linking of source programs, as well as loading the executable code into the hypercube nodes and initiating execution. Programs generally make no use of the SRM once they begin execution on the nodes.

3. Fine-Scale Parallel Integer Sort. The fine-scale parallel integer sorting algorithm is similar to that described in [10], however it makes use of the `send_to_queue` instruction [16] on the Connection Machine CM-2. This is a very powerful instruction that takes multiple messages for the same destination and stores them in a queue at the receiving processor. Each processor must have the same size buffer allocated to store the queue. This restriction is due to the SIMD nature of the Connection Machine, which employs a single stack pointer for processor memories and thus it is impossible to allocate variable amounts of memory across processors. The allocated buffer must also include a word in which to store the number of elements destined for the queue. If the buffer can store q_s messages, and some number greater than q_s of messages are sent to a particular processor, then the excess messages are lost but this word will still store the number of messages intended for that destination.

3.1. Fine-Scale Parallel "Queue-Sort" Algorithm. The n keys are stored in a one dimensional virtual processor (VP) set, call it VP1, of size n. Each VP has an index i and stores key K_i. The keys have range $[1, m]$, where m is no greater than O(n), therefore m buckets are needed to sort them. The main idea behind the algorithm is to create a queue for each bucket, perform a prefix sum over queue elements to compute the rank, and return the rank. The algorithm must be iterated when there are key densities greater than the maximum queue size. The steps in a single iteration of the *queue-sort* algorithm are as follows:

Queue-Sort Algorithm

1. In a distinct VP set, call it VP2, allocate memory in m virtual processors for a queue of size q_s. The value of q_s will depend on the available memory, in the analysis below we assume $mq_s = O(n)$.
2. Each processor in VP1 computes a destination address in VP2 based on the value of its key. The n processors of VP1 then collectively send their self-address to this destination using `send_to_queue`.
3. If this is the first iteration, then the m processors of VP2 collectively perform the prefix sum

   $$S_k = \sum_{i=0}^{k} |Q_i|$$

   where $|Q_i|$ indicates the number of elements in the queue for the i^{th} bucket, i.e., the key density. The result, for each bucket, is a sum, S_k, equal to the maximum rank for the keys in that bucket. Note that the value of $|Q_i|$ is known from execution of the `send_to_queue` instruction in the previous step.
4. The m processors of VP2 compute a rank for each queue member by subtracting each member's index (in the queue) from the maximum rank as given by S_k.
5. If more than q_s messages were sent to a queue, then S_k is adjusted as

   $$S_k \longleftarrow (S_k - q_s)$$

   in preparation for the next iteration.
6. The processors of VP2 send the computed ranks to the appropriate processors in VP1 as given by the addresses stored in each queue. Processors in VP1 which receive ranks are marked and do not participate in the next iteration.

Iterations are repeated until all the keys have been ranked.

3.2. Theoretical Analysis. Blelloch [5] describes a "scan-model" of computation for the Connection Machine (that is, the Exclusive Read Exclusive Write (EREW) model but including prefix of "scan" operations as unit-time primitives). This model is assumed in the following analysis.

The performance of this algorithm depends on the key density distribution. If ρ_{max} is the maximum key density, then $\lceil \rho_{max}/q_s \rceil$ iterations are required to complete the sort. Recall q_s is fixed by the available memory. Assume that $O(n)$ words of memory are available for m queue's, such that $q_s = O(n)/m$. In the following we will allow m to be any number less than but evenly divisible into n, yet greater than or equal to the number of physical processors, N_p. For example, m can be: $n/\log^2 n, n/\log n$ or n for $n = 2^{32}$. Therefore q_s will have size $\mathrm{O}(n/m)$. Clearly then, steps 1 and 5 will take $\mathrm{O}(1)$ time and step 4 will take $\mathrm{O}(n/m)$ time, each with $\mathrm{O}(m)$ processors. Communication is required in steps 2, 3 and 6; these require special consideration. In the scan model of the CM-2, step 3 requires $\mathrm{O}(1)$ time to complete using $\mathrm{O}(m)$ processors. The time for step 2 will depend on the number of combinations required to complete the `send_to_queue`. As is shown in the application analysis below, this is essentially given by the ratio n/m so step 2 has time complexity $\mathrm{O}(n/m)$. Step 6 must be carried out iteratively using at most q_s sub-iterations. The time for each sub-iteration is $\mathrm{O}(1)$ in the scan model, therefore the time to complete step 6 is $\mathrm{O}(n/m)$ using $\mathrm{O}(m)$ processors.

In summary, calculations carried out on VP1 have time complexity $T = O(n/m)$ and processor complexity $P = n$. Calculations carried out on VP2 have time complexity $T = O(n/m)$ and processor complexity $P = m$. Therefore one full iteration of *queue-sort* has parallel complexity (that is, PT) of $\mathrm{O}(n^2/m)$. The sort will complete in $\mathrm{O}(1)$ iterations only if ρ_{max} is $\mathrm{O}(n/m)$. This proves the following theorem:

THEOREM 3.1. *The queue-sort algorithm sorts a disordered sequence of n integers chosen randomly in the range [1, O(m)] with no more than O(n/m) repeated values in time O(n/m) using n scan model processors.*

3.3. Application Analysis. The amount of arithmetic computation in *queue-sort* is minimal, and the greatest contribution in time is from inter-processor communication. Inter-processor communication occurs only in steps 2, 3 and 6. Step 3 gets executed just once and has negligible effect on the overall execution time. Therefore only steps 2 and 6 need be considered for analysis. Since network contentions is reduced as fewer messages are transmitted, and since each successive iteration requires fewer messages to transmit, the time required by steps 2 and 6 decreases as the calculation proceeds. In the following, models are developed to account for the effect of network contention on communication performance.

For analysis, a disordered sequence of $n = 2^{23}$ keys was created in a sequential fashion on the front end and distributed uniformly over the physical processors of the Connection Machine. The key's were chosen from the range $[0,\ m)$ with an approximately Gaussian distribution and $m = 2^{19}$. Specifically, if r_i is a random fraction uniformly distributed over [0,1), then each key value was obtained as:

$$K_i \longleftarrow \lfloor m(r_{4i+0} + r_{4i+1} + r_{4i+2} + r_{4i+3})/4 \rfloor \quad \text{for} \quad i = 0, 1, \ldots, N-1.$$

A sample of the key density distribution is shown in figure 1. The maximum key density is actually 73, but the figure only shows key densities for every 128^{th} key value and the maximum is missed. There are 417,812 different keys.

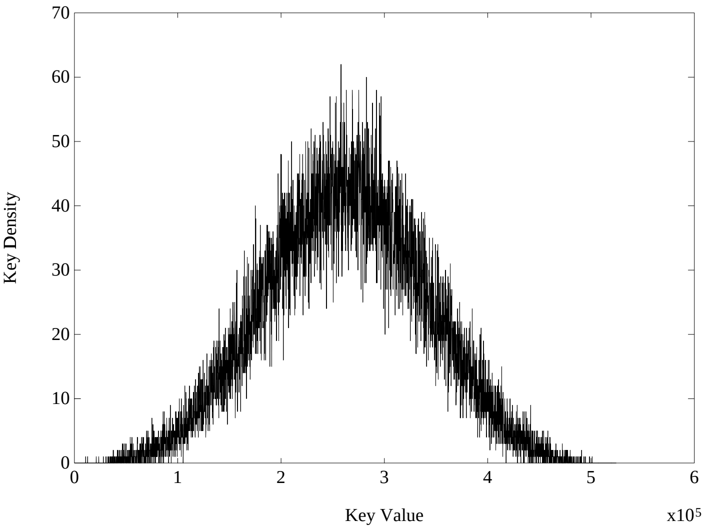

FIG. 1. *Sample of the approximate Gaussian key density distribution.*

As expected, steps 2 and 6 took the majority (≈97%) of the time. Figure 2 presents the time to sort, using 8k processors, as a function of the queue size. Note that the time spent in step 6 is independent of the queue size. Changing the queue size changes the number iterations necessary for queue-sort to complete. However, the total number of subiterations taken in step 6 always must equal ρ_{max}, for this reason its time is unaffected by the size of q_s. On the other hand, the time spent in step 2 *is* affected by the size of q_s. As q_s increases, fewer iterations are required so the overhead in using `send_to_queue` is paid fewer times. However, even when the number of iterations is constant, the time spent in `send_to_queue` decreases with increasing q_s. This implies that `send_to_queue` behaves in a manner similar to a conventional `send` in that the communication time is determined by the network bandwidth. The queueing of messages occurs in the network, so network contention has a great impact on the performance of `send_to_queue`.

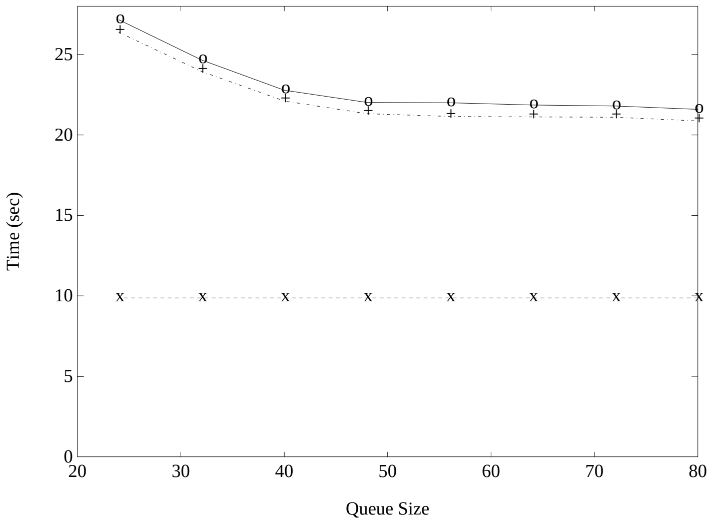

FIG. 2. *Time to sort as a function of queue size for Gaussian key distribution: —o— total time; -.-.+-.-. time for steps 2 and 6; - -x- - time for step 6.*

In the first iteration, all processors in VP1 are sending messages to VP2, therefore the time required by `send_to_queue` is constant regardless of q_s. However, in the second iteration, the number of active processors in VP1 decreases as q_s increases (because fewer keys remain to be ranked). Therefore the communication time decreases because of reduced network contention. Figure 3 presents the time accumulated by each `send_to_queue` instruction as q_s increases. It is evident from this figure that reducing network contention is more important than decreasing the communication start up cost in terms of improving performance. The issue of network contention is discussed more fully below.

Figure 4 presents the fraction of active processors in VP2 per subiteration of step 6. The values have been normalized by the total number of processors in the VP set. From this curve one can determine the number of messages communicated in a particular subiteration of step 6. Network contention affects step 6 in the same manner as step 2. As there are fewer keys remaining to be ranked, network traffic decreases and the communication time for each subiteration of step 6 decreases.

The solid curve in figure 5 presents the time spent per subiteration of

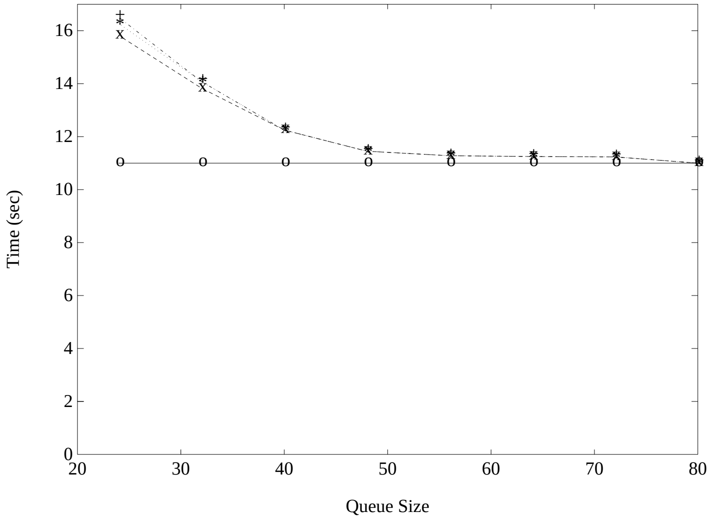

FIG. 3. *Time accumulated by each* `send_to_queue` *as a function of queue size: —o— first iteration; - -x- - second iteration; ···*··· third iteration; -.-.+-.-. fourth iteration.*

step 6 as a function of the natural logarithm of the fraction f of active processors in VP2. It has been normalized by the time spent in the first iteration (note that only 80% of the processors in VP2 were active in the first iteration). The curve is approximately linear for $\ln(f) > -4$, the abrupt deviation of the curve for $\ln(f) < -4$ occurs because the number of active virtual processors in VP2 drops below 8k, the number of physical processors. For purposes of analysis we can approximate this curve as

$$T_s = \Big(0.55 + 0.45\frac{1 + 0.25\ln f}{1 + 0.25\ln(0.8)}\Big)T_{s\text{o}} \tag{3.1}$$

where T_s is the total time, f is the fraction of processors active, and $T_{s\text{o}}$ is the time to send one message with $f = 0.8$. The experiment measured $T_\text{o} = 0.184$ sec with 8k processors, therefore

$$T_s = 0.189 + 0.0219\ln f. \tag{3.2}$$

The broken curve in figure 5 shows the model for T_s. Note that it is valid only while the number of active virtual processors is greater than the number of physical processors which, in this case, implies $f > 2^{-6}$. For smaller values of f we use $T_{s\text{o}} = 0.55T_{s\text{o}}$.

The same sort of approximation can be made for `send_to_queue` with the data from figure 3, however it is necessary to model the number of message collisions expected. This can be approximated by the ratio $R_{act} =$

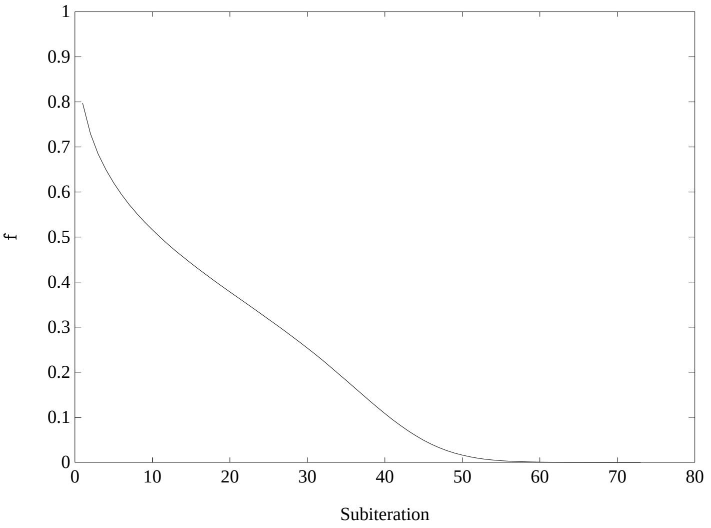

FIG. 4. *Fraction, f, of processors active in VP2 per subiteration of step 6 in the queue-sort algorithm for the Gaussian key density distribution.*

N_{vp1}/N_{vp2}, where N_{vp1} is the number of processors sending messages from VP1 and N_{vp2} is the number of processors receiving messages at VP2. Both these numbers can be obtained from figure 4. The solid line in figure 6 presents the the time measured for `send_to_queue` as a function of R_{act}. For $R_{act} > 4$, the curve is approximately linear, as expected. The abrupt deviation from linearity for $R_{act} < 4$ is due to both N_{vp1} and N_{vp2} dropping below N_p, the number of physical processors. One can model `send_to_queue` by

$$T_q = (R_{act} - R_{act_\circ})T_{q\circ} \tag{3.3}$$

where $R_{act_\circ}$ is the value of R_{act} where N_{vp1} and N_{vp2} drop below N_p, and $T_{q\circ}$ is chosen to give the correct time for the initial value of R_{act}. For the Gaussian key density distribution, the first iteration had $R_{act} = 20.1$ and the measured time for `send_to_queue` was 11.0 sec, thus $T_{q\circ} = 0.68$ sec with $R_{act_\circ} = 4.0$. The broken curve in figure 5 shows our model for T_q. Note that it is valid only while $R_{act} > 4.0$, and for smaller values the model uses $T_q = 0.24$, the time measured for $N_{vp1} = N_{vp2} = 1$.

In order to test these models, queue-sort was applied to a disordered sequence of integer keys sampled from the range $[0,\ m)$ this time with a linear distribution. Specifically, key values were assigned as

$$K_i \longleftarrow m\sqrt{r_i}$$

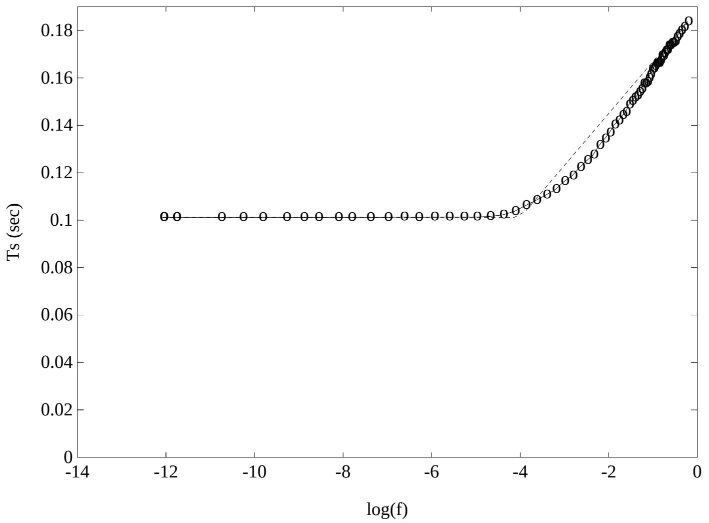

FIG. 5. *Time per subiteration of step 6 as function of* $\ln f$ *for the Gaussian key density distribution: —o— measured; - - - predicted by model.*

with $m = 2^{19}$. A sample of the key density distribution is shown in figure 7; the maximum key density was 58 and there were 507810 different keys. Figure 8 presents the number of active processors in VP2 per subiteration of step 6. Using figure 8, the model for T_s predicts a time of 8.1 sec to complete step 6. The measured time was 7.7 sec, which is within 5% of our predicted time. Figure 9 presents the measured and predicted times for T_q as a function of R_{act}. The agreement is very close especially for large values of R_{act}, which, of course, is how the model was calibrated. Finally, figure 10 presents the measured and predicted times for queue-sort for the density distribution of figure 7. Again there is excellent agreement, with the model being accurate to 5% of the measured times.

It should be obvious from these results that the effect of decreasing network contention on communication performance must be considered when analyzing communication-iterative algorithms like queue-sort. The success of these models in predicting communication performance as a function of network contention is extremely encouraging and indicates that the performance of complex and changing communication patterns can be predicted with some accuracy using relatively simple models for communication.

4. Medium-Scale Parallel Integer Sort. The medium-scale parallel integer sorting algorithm attempts to load balance the sorting problem through an approximate representation of the key density distribution. Typ-

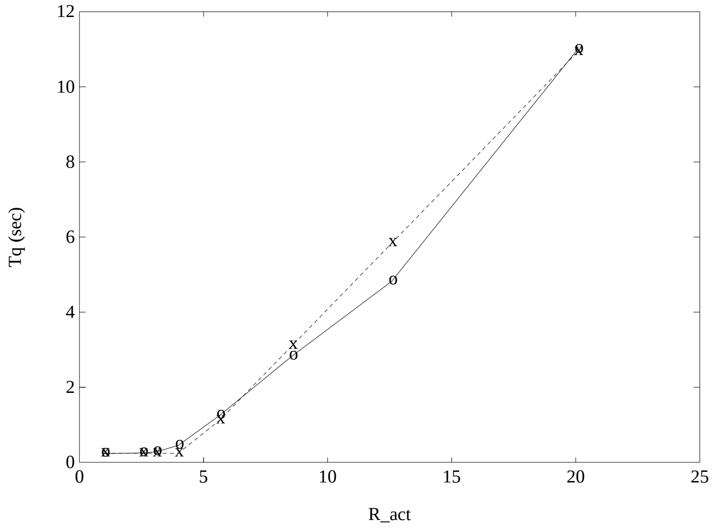

FIG. 6. *Time for* `send_to_queue` *as function of R_{act}, the ratio of active processors in VP1 to active processors in VP2, for the Gaussian key density distribution: —o— measured; - -x- - predicted by model.*

ically, in a sequential bucket sort there are m buckets for m possible key values. The parallel algorithm, however, uses m' *barrels* (where $m' < m$) to determine the number of keys in each m/m' subrange of key values. The counts stored in the barrels are used to approximate the expected loads for each processor and thereby determine a balanced decomposition. The idea shares some similarities to that of sampling the sequence to obtain a balanced decomposition, although the latter has been applied primarily for the case of general comparison based sorting. Huang and Chow [12] first proposed sampling the sequence as a means of determining an appropriate partitioning for the data. Shi and Schaeffer [15] apply the same idea with great success in their Parallel Sorting by Regular Sampling (PSRS) algorithm. In PSRS, a regular sample taken from the (locally ordered) sequence is globally sorted and itself gets sampled to determine the parallel decomposition used in the final global ordering of the sequence. Barrel sort differs in that it uses information from the whole sequence, not just a sample, to determine a suitable parallel decomposition.

4.1. Medium-Scale Parallel "Barrel-Sort" Algorithm. The n keys are evenly distributed across p processors such that each processor, k ($k \in [1,p]$) has keys $\{K_i | i \in [(k-1)n/p, kn/p)\}$. In the following, the notation $\{\cdot\}$ will be used to indicate a sequence of n elements distributed in some unspecified manner over p processors. The keys have range [1,m], where m

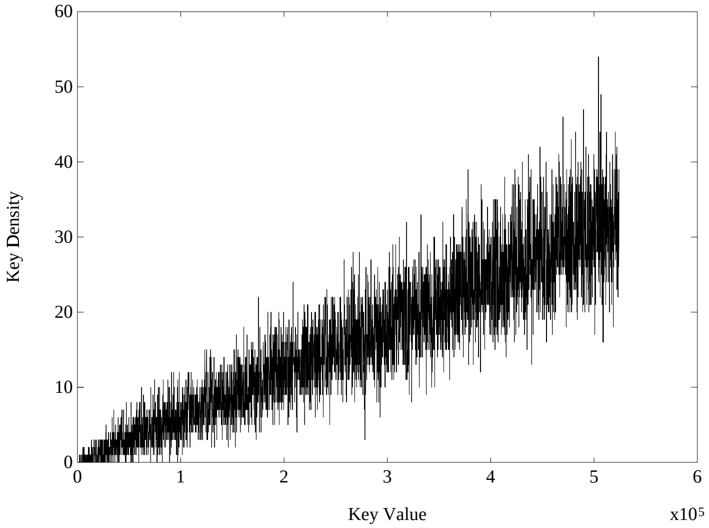

FIG. 7. *Sample of the linear key density distribution.*

is no greater than O(n), therefore m buckets are needed to sort them. Since it is impossible for each processor to keep m buckets, the processors have to work on some subrange of key values. The main idea behind the *barrel-sort* algorithm is to use a smaller number, m', of barrels to determine how to distribute the m buckets. The size of m' will depend on the available memory. The steps in the algorithm are as follows:

Barrel-Sort Algorithm

1. Each processor counts the number of keys falling into each of its m' barrels. Let B_{jk} be the j^{th} barrel in processor k. Each barrel, B_{jk}, will store the count of keys in the subrange $[jm/m', (j+1)m/m')$ in processor k.
2. Compute the sum $B_j = \sum_{k=1}^{p} B_{jk}$, and copy the result to all processors.
3. Use B_j to distribute subranges of $[1,m]$ such that each processor will receive approximately the same number of keys. For a perfect load balance, each processor, k, should handle a subrange $[a_k, b_k)$ with n/p keys. Values of a_k, b_k that approximately yield this load distribution are determined from B_j as follows:
   (a) Perform a prefix sum on B_j. Let the result be S_j.
   (b) In all processors, for $k = 1, \ldots, p$, find the index l_k in S_j such that $S_{l_k} \leq kn/p < S_{l_k+1}$.
   (c) Set $b_k = l_k m/m'$.
   (d) Set $a_1 = 1$, and $a_k = b_{k-1}$ for $k > 1$.

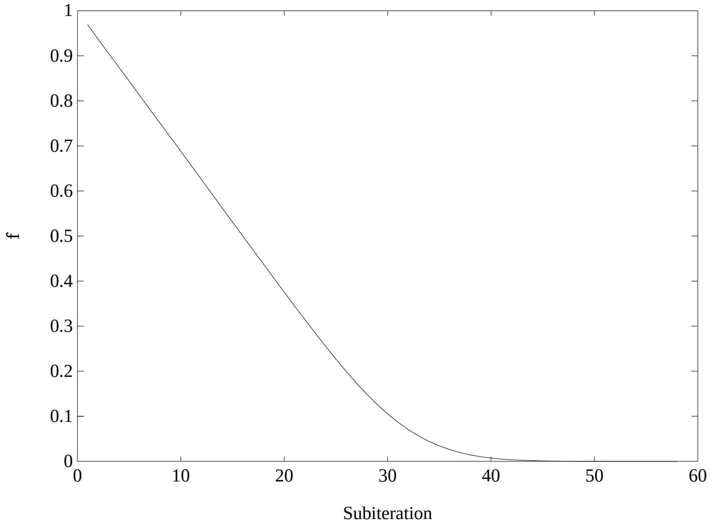

FIG. 8. *Fraction, f, of processors active in VP2 per subiteration of step 6 in the queue-sort algorithm with the linear key density distribution.*

4. Each processor determines which subrange $[a_k, b_k)$ contains each of its keys and stores the result in a local array P_i. Each value in P_i is a pointer to the processor whose assigned subrange of buckets contains key value K_i. This step requires a binary search in $\{(a_k, b_k)|k \in [1,p]\}$.
5. Each processor sends to all other processors the keys in that other processor's subrange. This is carried out by having each processor rank its list of keys according to P_i and permute the key and index sequences, $\{K_i\}$ and $\{i\}$, accordingly. Let $\{K_q\}$ be the sorted $\{K_i\}$ and let $\{I_q\}$ be the index in $\{K_i\}$ for $\{K_q\}$. Note that P_i has range [1,p] and sorting is carried out strictly local to each processor. Each processor then sends the appropriate subsequence of $\{K_q\}$ and $\{I_q\}$ to the corresponding processor. This is an all-to-all (or complete exchange) type of communication with message lengths of varying sizes which permutes $\{K_q\}$ into a new sequence $\{K_r\}$. At the end of this step, every processor k stores a subsequence of $\{K_r\}$, approximately of length n/p, and the corresponding subsequence of $\{I_r\}$, where $\{I_r\}$ are the indices in $\{K_i\}$ for the keys in $\{K_r\}$. Furthermore, each subsequence of $\{K_r\}$ has range $[a_k, b_k)$.
6. Each processor ranks its subsequence of keys and permutes its subsequence of $\{I_r\}$ accordingly. Let $\{I_s\}$ be this permuted sequence. Permuting $\{K_r\}$ at this point would result in a sorted sequence of

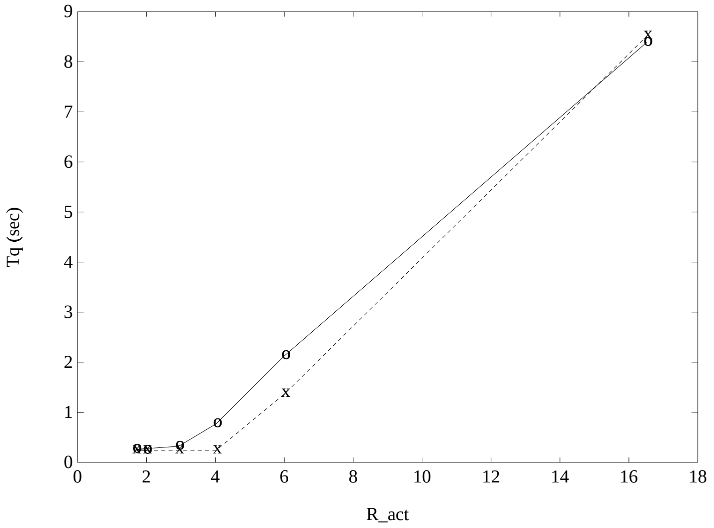

Fig. 9. *Time for* `send_to_queue` *as function of R_{act}, the ratio of active processors in VP1 to active processors in VP2, with the linear key density distribution: —o— measured; - -x- - predicted by model.*

keys. However, the objective is not to sort the original sequence of keys but rather to find the permutation which sorts it (under the assumption that the records associated with the keys are large and one wants to permute them just once). Nonetheless, assume such a permutation was carried out, then $\{K_s\}$ would be the sorted sequence of $\{K_i\}$, and $\{I_s\}$ would be the index in $\{K_i\}$ for $\{K_s\}$. Therefore the permutation, $\{R_i\}$, which converts $\{K_i\}$ into $\{K_s\}$ is computed as

$$R(I_s) \longleftarrow s.$$

This is carried out as follows:

(a) From $\{I_s\}$, create an array of pointers P_s pointing to the processor which stores $K_i(I_s)$ in the original sequence $\{K_i\}$.

(b) Use P_s to create $p-1$ buffers in each processor to store the local values of $\{I_s\}$ and $\{s\}$ which need to be sent to other processors. Where P_s points to the local processor, compute the rank $R(I_s)$ as described above.

(c) Each processor sends its $p-1$ buffers to the $p-1$ other processors. This is an all-to-all type of communication with message lengths of varying sizes. At the end of this step each processor has the values of I_s and s corresponding to its subsequence of

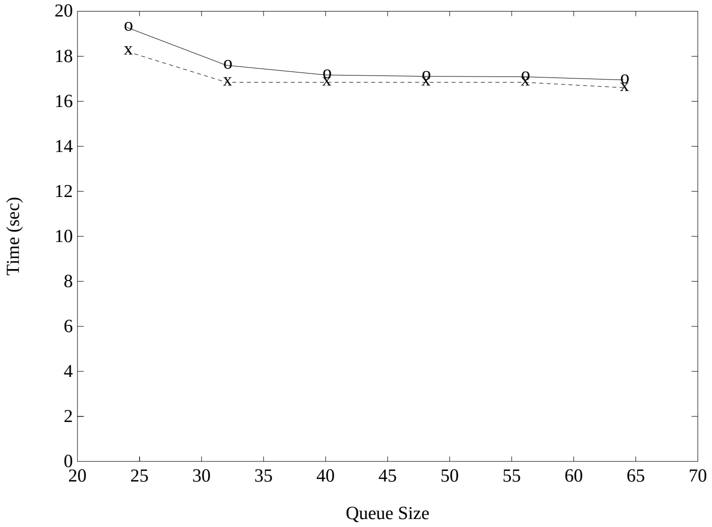

FIG. 10. *Time to complete queue-sort as function of queue size with the linear key density distribution: —o— measured; - -x- - predicted by models.*

$\{K_i\}$.

(d) Each processor computes ranks $R(I_s)$ with its received values of I_s and s.

4.2. Theoretical Analysis. The following analysis is based on the distributed memory MIMD model presented by the Intel iPSC/860. Note that this model is weaker than either the scan model or the strict EREW model since it does not allow exclusive reads from parallel memory. Parallel memory access must be initiated by a `send` instruction and is completed only by a `receive` instruction. This model here will be referred to as the message passing model.

For purposes of analysis, assume $m' = p^2$. Each processor stores n/p keys, so step 1 takes $O(n/p)$ time and steps 2 and 3 take $O(p)$ and $O(p^2)$ time respectively. Step 4 requires each processor to perform n/p local binary searches over p elements and thus takes $O(n/p \log p)$ time. Step 5 requires each processor to locally sort n/p elements in the range $[1, p]$. This can be carried out with sequential bucket sort in $O(n/p)$ time. Finally, the time for step 6 will depend on how successful step 3 was in its decomposition of the key range. Assuming that each processor has $O(n/p)$ keys, then step 6 will require $O(n/p)$ time. Note that step 6a does *not* require searching because $\{K_i\}$ is evenly distributed across the processors.

It is conceivable for there to be more than $O(n/p)$ keys in a processor in step 6. Such a situation would arise if the key density distribution had a large number of repeated keys. Specifically, there would have to be at least one barrel with greater than O(n/p) keys. Since each barrel accounts for m/p^2 buckets, this implies there should be no more than O(n/p) repeated keys in each m/p^2 subrange of key values. This proves the following theorem:

THEOREM 4.1. *The barrel-sort algorithm sorts a disordered sequence of n integers chosen randomly in the range [1, O(n)] with no more than O(n/p) repeated values per O(n/p^2) subrange in time O($n/p \log p$) using p message passing processors.*

4.3. Application Analysis. Unlike queue-sort, the amount of arithmetic computation in barrel-sort can be a substantial part of the calculation. The difference is due to the medium-grain parallelism targeted by barrel-sort. The major contributions from arithmetic computation occur in steps 1, 4, 5 and 6. Inter-processor communication takes place in steps 2, 5 and 6.

The Gaussian-distributed disordered sequence used to analyze queue-sort was also used with barrel sort (see figure 1). With $m = 2^{19}$, $m' = 2^{11}$ and $n = 2^{23}$, using all 128 processors the time to complete barrel-sort was 3.20 sec. The profiling was carried out with 64 processors for which the total time was 5.25 sec. Of this, step 2 required 0.10 sec, and the communication in steps 5 and 6 required 1.08 and 0.69 sec respectively. The remaining 3.38 sec were due to computation.

Communication time, T_{comm}, on the iPSC/860 is adequately modelled by

$$T_{comm}(k) = t_{\mathrm{o}} + k t_{send} \tag{4.1}$$

where k is the number of bytes in the message, t_{o} is the latency, and t_{send} is the time per byte. Using the numbers from [8], for long messages (greater than 100 bytes), $t_{\mathrm{o}} = 149\mu$ sec and $t_{send} = 0.36\mu$ sec and for short messages $t_{\mathrm{o}_{sh}} = 74\mu$ sec and $t_{send_{sh}} = 0.19\mu$ sec. Step 2 can be implemented using a pairwise exchange algorithm such that only $\log p$ messages are required per processor, each of length $4m'$ bytes. Each complete exchange in steps 5 and 6 is implemented as $3(p-1)$ messages per processor, with $(p-1)$ messages of length 4 bytes and $2(p-1)$ messages approximately of length $4n/p^2$ bytes. The actual length depends on decomposition determined in step 3; for the test case all the message lengths were within 9% of this value. Taken altogether, interprocessor communication takes

$$T_{comm} = [4(p-1) + \log p] t_{\mathrm{o}} + 2(p-1) t_{\mathrm{o}_{sh}} + 4[4(p-1)n/p^2 + m' \log p] t_{send} + 8(p-1) t_{send_{sh}}. \tag{4.2}$$

For 64 processors and sequence parameters above, (5) works out to 0.81 sec. The measured time was 1.87 sec. The difference between the expected and

TABLE 1
Performance of queue-sort and barrel-sort.

Algorithm	Processors	time (sec) Gaussian	time (sec) Linear
queue-sort	8k	21.51	16.80
(on CM-2)	16k	11.15	8.60
	32k	5.60	4.38
barrel-sort	64	5.25	5.53
(on iPSC)	128	3.20	3.52
bucket-sort	1	1.15	1.15
(on YMP)	8	0.19	0.19

measured time is attributable to an idiosyncrasy in the iPSC communication hardware wherein a send and receive occurring a short interval apart are carried out sequentially when they could be carried out concurrently. This effectively doubles the communication time in a complete exchange, bringing the predicted time up to 1.58 sec. Furthermore, accounting for the longest messages in each step of the exchange (which amounted to a total transmission 7% greater than expected) results in a predicted time of 1.68 sec, which is within 10% of the measured time. Bokhari [7] describes a complete exchange algorithm on the iPSC which does not suffer from this idiosyncrasy and would conceivably result in much improved communication performance.

5. Discussion. Table 1 presents the best results for queue-sort and barrel-sort on the Connection Machine CM-2 and the iPSC/860 respectively. Times are given for both the Gaussian distributed and the linear distributed key densities (see figures 1 and 7). The maximum queue size in queue-sort was made large enough to allow completion in a single iteration. In a real application, the memory available for the queues would be limited to that which was unused by the rest of the application, the resulting impact on performance may be deduced from figure 2. The number of barrels used in barrel-sort was 2048. The current implementation of barrel-sort could not be run with less than 64 processors because of memory restrictions, although with some modifications 32 processor results could be obtained. For comparison, the performance of a vectorized bucket sort on 1 and 8 processors of the Cray YMP is also given. The Cray YMP code was obtained from Cray Research Inc. in response to the NAS Parallel Benchmarks and is highly tuned to this architecture.

It is encouraging to see that the performance of queue-sort and barrel-sort are comparable. Queue-sort involves virtually no arithmetic computation but depends on many single-word transmissions to order the data. The total amount of data motion is about the same for both queue-sort and barrel-sort. Queue-sort essentially consists of n single-word transmissions using n processors followed by n single-word transmissions using n/m processors. Therefore for queue-sort to be competitive the overhead on message transmission must be very low. On the other hand, barrel-sort consists of two complete exchanges each involving $p-1$ transmissions of approximately n/p^2 words using p processors. Barrel-sort attempts to minimize the number of messages transmitted at the expense of additional arithmetic computation. Therefore barrel-sort should perform well on machines with a high overhead on message transmission so long as medium-scale parallelism is available.

The Cray YMP performance taken from [4] is given for comparison. Neither queue-sort on the CM-2 nor barrel-sort on the iPSC/860 can match the performance of a vectorized bucket-sort on the YMP. In general, sorting requires a very high memory bandwidth and relatively little computation. The high memory bandwidth is a well known feature of the Cray machines and one expects good performance on the YMP for this problem. These parallel architectures have exhibited comparatively good performance for this problem with highly tuned radix sorts (see [2, 4, 6]), however the results presented in this paper are intended primarily for understanding the algorithms, not benchmarking the machines. Benchmark results for this problem may be found in [4].

6. Conclusions. Two new parallel integer sorting algorithms, queue-sort and barrel-sort, have been presented and analyzed in detail. These algorithms do not have optimal parallel complexity, yet they show respectable performance in practice. Queue-sort is designed for fine-scale parallel architectures which allow the queueing of multiple messages to the same destination. Barrel-sort is designed for medium-scale parallel architectures with a high message passing overhead. The performance results from the implementation of queue-sort on a Connection Machine and barrel-sort on a 128 processor iPSC/860 are presented and compared to a a fully vectorized bucket sort on the Cray YMP. The parallel machines show poorer performance because of their comparatively slower memory systems.

Acknowledgements. This work was supported through NASA Contract NAS 2-12961.

REFERENCES

[1] Aho, A.V., Hopcroft, J.E., Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., 1974.

[2] BABER, M., *An Implementation of the Radix Sorting Algorithm on the Touchstone Delta Prototype,* 6th Distributed Computing Conference Proceedings, Portland, OR, April 28 - May 1, 1991.

[3] BAILEY, D.H., BARTON, J., LASINSKI, T., AND SIMON, H., *The NAS Parallel Benchmarks*, Technical Report RNR-91-02, NASA Ames Research Center, Moffett Field, CA 94035, January 1991.

[4] BAILEY, D.H., BARSZCZ, E., DAGUM, L., AND SIMON, H.D., *NAS Parallel Benchmark Results*, Supercomputing '92, Minneapolis, MN, Nov 16-20, 1992.

[5] BLELLOCH, G., *Scans as Primitive Parallel Operations*, Proceedings of 1987 Int. Conf. on Parallel Processing, University Park, PA, 1987.

[6] BLELLOCH, G.E., LEISERSON, C.E., MAGGS, B.M., PLAXTON, C.G., SMITH, S.J., AND ZAGHA, M., *A Comparison of Sorting Algorithms for the Connection Machine CM-2,* 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, July 21-24, 1991.

[7] BOKHARI, S.H., *Complete Exchange on the iPSC-860,* ICASE Report No. 91-4, NASA Langley Research Center, Hampton, VA 23665, January 1991.

[8] BARSZCZ, E., *One Year with an iPSC/860*, Technical Report RNR-91-001, NASA Ames Research Center, Moffett Field, CA, January 1991.

[9] CHLEBUS, B.S., *Parallel Iterated Bucket Sort*, Information Processing Letters, vol. 31, no. 4, pp. 181-183, 1989.

[10] DAGUM, L., *On the Suitability of the Connection Machine for Direct Particle Simulation*, Ph.D. Thesis, Stanford University, Stanford, CA, June 1990.

[11] DAGUM, L., *Sorting for Particle Flow Simulation on the Connection Machine,* Concurrency: Practice and Experience, vol. 4(3), pp. 241-255, May 1992.

[12] HUANG, J.S. AND CHOW, Y.C., *Parallel Sorting and Data Partitioning by Sampling,* COMPSAC 83, pp. 627-631, Chicago IL, Nov 7-11, 1983.

[13] KNUTH, D.E., *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Publishing Co., Menlo Park, 1973.

[14] RAJASEKARAN, S., REIF, J.H., *Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms*, SIAM J. Comput., Vol. 18, No. 3, pp. 594-607, 1989.

[15] SHI, H., SCHAEFFER, J., *Parallel Sorting by Regular Sampling*, Journal of Parallel and Distributed Computing, Vol. 14, pp. 361-372, 1992.

[16] THINKING MACHINES CORP., *The Connection Machine System: Paris Reference Manual, Version 6.0*, Thinking Machines Corp., Cambridge MA, 1990.